Introduction to Programming

Block Tutorial C/C++

Michael Bader

Master's Program
"Computational Science and Engineering"

C/C++ Tutorial – Overview

- From Maple to C
- Variables, Operators, Statements
- Functions: declaration, definition, parameters
- Arrays and Pointers
- From Pointers to Dynamical Data Structures
- Object Oriented Programming: From C to C++

Friday Lesson – Overview

- Object Oriented Programming
- Objects and Classes in C++
- Information Hiding
- Inheritance
- Polymorphism

What is an Object?

Definition:
An object is a software bundle of variables and related methods.

An object is characterized by its identity, its state, and its behaviour:

- the **identity** of an object distinguishes it from all other objects;
- the **state** of an object is given by its variables;
- **behaviour** of an object is specified by its methods (functions, procedures)

Objects can reflect real world components that are part of the problem.

What is a Class?

Definition:
A class is a blueprint that defines the variables and the methods common to all objects of a certain kind.

Objects vs. Classes:

- objects are **instances** of a class
- classes are implemented by the programmer – objects are created during the execution of a program
- all objects have the same behaviour (i.e. methods); methods are specified for an entire class
- all objects of a class have the same variables ("members"), but with different values ("state")

Paradigms in Object Oriented Programming

Abstraction

Encapsulation

Modularity

Inheritance

Polymorphism

What is Abstraction?

Abstraction means focusing on the essential properties of an object.

- ignore details
- ask "what can an object do?" instead of "how is it going to do it?"
- top-down approach
- use objects and classes for program design (instead of memory cells)
- specify interfaces before starting the implementation

Abstraction is part of the object oriented **design process**.

What is Encapsulation?

Encapsulation is also known as **Information Hiding**:

Implementational issues of a class should be hidden from third parties as far as possible.

- access to `private` members or functions of another class is forbidden
- modifying the state of an object is only possible using `public` methods (and members);
  the public members (and methods) define the **interface** of a class

Goal: All information and processes associated with an object should be contained within its definition.

Consequence: implementation of a class can be changed without affecting its behaviour.

What is Modularity?

Modularity means partitioning a program (project) into smaller units, each performing a well designed task.

Goals:

- group related functions/procedures/objects
- well-defined interfaces between modules
- division of responsibilities
- make application code more readable

Consequence: different programmers or teams can work on different modules of a project.

What is Inheritance?

Inheritance means that a class can inherit state and behaviour from a superclass.

Goals:

- re-use of existing (error free) code
- define generic behaviour of a superclass that can be inherited by several subclasses

Example: Vehicles (= superclass)

- possible subclasses: car, airplane, horse cart, truck, bicycle
- subclasses of cars: limousine, convertible, etc.

Inheritance vs. Composition

Inheritance: "is-a"-relationship

Example: Vector "is-a" Array of Integer (?)

- access of elements identical
- mathematical operations may be missing

Composition: "has-a"-/"consists-of"-relationship

Example: Vector "consists-of" Array of Integer (?)

- Assume: Array is resizable $\Rightarrow$ Vector resizable?
- different range of indices?

$\Rightarrow$ **Sometimes a Difficult Decision!**

What is Polymorphism?

- different methods sharing the same name can be defined for different classes
- pick the method that corresponds to the type of the object
- corresponding method is chosen **at runtime**
- with inheritance, this even works if the syntax (at compile-time) suggests another method

$\Rightarrow$ Define different implementations for common interfaces

Polymorphism and Interfaces

Interface: "abstract" superclass that determines common behaviour (methods) of its subclasses

Example: different matrix implementations

- specify set of interface methods: `getElem`, `setElem`, `multiply`
- implement methods in subclasses (`SparseMatrix`, `TriMatrix`, . . .)
- solvers stick to interface methods
- required matrix methods are chosen at runtime depending on which type of matrix implementation is used

What is an "Object Oriented Language"?

OO Programming/Design: technique/approach how to design and implement software

OO Language: programming language that **supports** OO programming

Object oriented programming languages should

$\Rightarrow$ support classes/objects

$\Rightarrow$ support inheritance/polymorphism

$\Rightarrow$ support information hiding/encapsulation

"support" means more than just "it is possible to do it"!

Classes in C++

Example: a class definition for vectors

```
class Vector {
  public:                 // define methods:
   Vector(int s);         // constructor
   ~Vector();             // destructor

   double getElem(int index);
   void setElem(int index, double value);

  private:                // define member variables:
   int size;              // size of the Vector
   double* elems;         // array of elements
}
```

Public and Private Members/Methods

C++ allows to place certain access restrictions on its members and methods:

- `public` members or methods can be used by all functions or methods to modify an object's state
- `private` members or methods can only be accessed by methods of the same class. Other methods or functions cannot access or modify private members.
- `protected` members or methods can also be accessed by methods of subclasses (see *inheritance*).

⇒ **Information Hiding, Encapsulation**

Creating and Deleting Objects

Constructors and **destructors** are methods that create and delete objects of a class:

- initialize member variables
- allocate/delete memory for required dynamical data structures

In C++, constructors and destructors

- do not have any return type
- have the same name as the class

Constructors can have parameters, destructors never have any parameters.

Constructors and Destructors

Example:

```
Vector::Vector(int s) {
   size = s;
   elems = new double[s];
   for(int i=0;i<size;i++)
      elems[i] = 0.0;  // initialize elements
}

Vector::~Vector() {
   delete elems;
}
```

Method definitions are to be placed outside the class definition

`new` and `delete`

The C++ operators `new` and `delete` create and delete objects dynamically. `new` (like `malloc` in C) returns a pointer to the created object.

Example:

```
Vector* v;
v = new Vector(5); // define a new vector of 5 elements
delete v;
```

`new` can also be used to create arrays.

Example:

```
double* elems;
elems = new double[10]; // define a new vector of 5 elements
```

Definition of Member Functions (Methods)

Example:

```
double Vector::getElem(int index) {
   if ( index>=0 && index<size )
      return elems[index];
   cerr << "Array Index out of Bounds!\n"; //error!
}

void Vector::setElem(int index, double value) {
   if ( index>=0 && index<size ) {
      elems[index] = value;
      return;
   }
   cerr << "Array Index out of Bounds!\n";
}
```

Calling Member Functions (Methods)

Example:

```
main() {
   int i;
   Vector v(10),w(10);
   for(i=0;i<10;i++) {
      v.setElem(i,i); w.setElem(i,10.0-i); }
   for(i=0;i<10;i++)
      cout << v.getElem(i) << ' ' << w.getElem(i) << endl;
}
```

- calls refer to objects
- members of the called object are "additional parameters"

Inheritance in C++

Example: class definition for matrix lines

```
class MatrixLine : public Vector {
   public:
      MatrixLine(int s); // Constructors are never inherited
      ~MatrixLine();

   double multiply(Vector c);
}
```

`MatrixLines` inherits from class `Vector`:

- methods `getElem` and `setElem`
- members `size` and `elems`

Polymorphism and Interfaces in C++

Example: matrix interface

```
class Matrix {
   public:
      virtual double getElem(int, int) = 0;
      virtual void setElem(int, int, double) = 0;
      virtual Vector& multiply(const Vector&) = 0;
}
class SparseMatrix : public Matrix {
   public:
      virtual double getElem(int, int);
      virtual void setElem(int, int, double);
      virtual Vector& multiply(const Vector&);
      ...
}
```

Polymorphism and Interfaces in C++ (2)

Example:

```
main() {
    Matrix *m;
    SparseMatrix sm(10);
    TriMatrix tm(10);
    Vector v(10);
    ...
    m = &sm; m->multiply(v);
    m = &tm; m->multiply(v);
}
```

The decision which implementation of `multiply` has to be used is taken at runtime.

Features of C++ that are not "Object Oriented"

- overloading of functions:

```
Vector(int size);
Vector(int size, double initValue);
```

- overloading of operators:

```
Matrix& operator*=(const Matrix& otherMatrix);
```

- `new` and `delete`
- Reference types: `Matrix &m`, i.e. "pointers without pointer arithmetics"
- strict type checking, inline functions, templates, exceptions, . . .